Building Applications with VTK Library Using Tcl, C++ and Fortran Components

Roman Putanowicz, Frédéric Magoulès

Building Applications with VTK Library Using Tcl, C++ and Fortran Components

Roman Putanowicz
<putanowr@twins.pk.edu.pl>
Frédéric Magoulès
<frederic.magoules@iecn.u-nancy.fr>

Revision : 1.5
February 23, 2003

Contents

1 Introduction

In programs using VTK library, visualization process can be described in terms of data flow through so called *visualization network* or *visualization pipeline* [1, 2].

During visualization, a data, which is represented by *visualization objects*, is passed between *process objects* connected into the visualization pipeline. The process objects operate on input data to generate output data.

The process objects can be divided into following categories: *source objects, filter objects* and *mapper objects*. Filter objects require one or more input data objects and generate one or more output data objects. VTK provides several filter objects which perform various visualization operations (e.g. extracting geometry, extracting and modifying data attributes, etc).

2 Programmable filters

When new processing capabilities are required, and they cannot be obtained by combination of existing filters, new classes of filters can be added. This requires introduction of a new class and modification of the source code. However it is possible to create new type filter objects without creating new classes and even to create new kind of filters on the run time and from the scripting languages like Tcl or Python. To make it possible VTK provides family of programmable filter class which has all common properties of ordinary filter classes except the fact that their processing routine can be set to specified user function. This way the user has only to write the processing function, create new instance of programmable filter and use it to build visualization pipeline. Each time the filter is requested to execute, it will call user specified function. The next section show how to write functions for programmable filters (`vtkProgrammableFilter` in particular) in Tcl, C++ and Fortran.

3 Accessing VTK object's data

It might happen that we want to extend visualization programs by functions written in Fortran or we have large Fortran legacy code we would like to interface with visualization program written using VTK library. One problem that immediately appears is that in Fortran (including Fortran 90) we do not have direct access to C++ objects. In theory it is possible to pass C++ object pointer to a Fortran function and then knowing the memory layout of the object manipulate it directly from Fortran but this is restricted to simplest cases and is highly not portable. What instead should be done is to extract all necessary information from C++ object, pack it into ordinary variables and arrays and pass that data to a Fortran function. When Fortran function returns modified arguments, they are used to alter the C++ objects or to create new ones.

We assume that the reader is already familiar with basic VTK components and in particular with VTK data model. If not then we suggest reading chapters 4 and 5 from [1] or chapter 11 from [2]. Nevertheless we will start our discussion with a very simple example which introduces one of the VTK array classes – `vtkDoubleArray`

3.1 Creating and manipulating VTK arrays

The example below shows how to create an array of double values with 10 rows and 3 columns. Such array could be used for instance to hold points coordinates.

```
1  #include <iostream>
2  #include "vtkDoubleArray.h"

3  using namespace std;

4  int main(void)
5  {
6    vtkDoubleArray *array;
7    int m = 10;
8    int n = 3;
9    double buff[3];

10   /* Creating VTK array object */
11   array = vtkDoubleArray::New();
12   array->SetNumberOfComponents(n);

13   array->Allocate(m);

14   array->SetNumberOfTuples(m);

15   for (int i=0; i<m; i++)
16   {
17     buff[0] = buff[1] = buff[2] = (double)i;
18     array->SetTuple(i, buff);
19   }

20   /* copy values from 'array' object to the 'carray' array */
21   int nrows = array->GetNumberOfTuples();
22   int ncols = array->GetNumberOfComponents();
23   double *carray = new double [nrows * ncols];

24   for (int i=0; i<nrows; i++)
25   {
26     for (int j=0; j<ncols; j++)
27     {
28       carray[ncols*i + j] = array->GetValue(ncols*i+j);
29     }
30   }

31   /* print the 'carray' out */
32   for (int i=0; i<nrows; i++)
33   {
34     for (int j=0; j<ncols; j++)
35     {
36       cout <<  " " << carray[ncols*i + j];
37     }
38     cout << "\n";
39   }

40   delete [] carray;
41   array->Delete();
```

```
42   return 0;
43 }
```

In the lines 11 to 14 we create new array object, set number of components (number of columns) and allocate memory enough to hold 10 rows (tuples). Then we set actual number of tuples to be 10.

In lines 15 to 19 we initialize the array.

Lines 21 to 23 create one dimensional "C" array big enough to hold all values from the 'array' object. When interfacing with Fortran the operation of copying data from VTK object to one dimensional array and to initializing VTK object from an array will be so frequent that it pays of to create a library which covers the most common cases.

3.2 Data passing library

As we said in the previous section it could be beneficial to create special functions to facilitate data transfer from and to VTK objects. Considering the example above we could introduce the following function

```
1     double *dplGetDoubleCArray(vtkDoubleArray *array, int *length=NULL)
2     {
3        assert (array!=NULL);
4        int maxId = array->GetMaxId();
5        // in C/C++ indexing starts from 0
6        double *carray = new double[maxId+1];
7        if (carray != NULL)
8        {
9          for (int i=0; i<=maxId; i++)
10         {
11           carray[i] = array->GetValue(i);
12         }
13         if (length != NULL)
14         {
15           length = maxId+1;
16         }
17       }
18       return carray;
19    }
```

In appendix A we present more functions which can help to transfer data from and to VTK objects. Those functions are organized into small library called `dpl` – data passing library. That library covers only few cases of accessing VTK objects – in particular following classes: `vktUnstructuredGrid, vtkPoints, vtkPointData`. If necessary, more robust and universal functions can be easily written based on the `dpl` examples.

3.3 Passing data from VTK object to Fortran routine

To finish this section we present another example of accessing VTK object. This time we use `dpl` functions to extract values of the points scalar attribute called "tempera-

ture", pack it into an array and send that array together with its length to a Fortran function which substitutes each element with its sinus value.

```
1  #inlcude "dpl.h"
2  #include "vtkProgrammableFilter.h"
3  #include "vtkUnstructuredGrid.h"
4  #include "vtkDataSet.h"

5  void SinusTemperature(void *arg)
6  {
7    vtkProgrammableAttributeDataFilter *myFilter;
8    vtkDataSet *input;
9    vktPointData *pd;
10   double *indata;
11   int length;

12   *myFilter = (vtkProgrammableAttributeDataFilter *)arg;
13   input= myFilter->Input();
14   pd = input->GetPointData();

15   /* get the point data array */
16   indata = dplGetScalarsCArray(pd,  "temperature", &length);
17   if (indata != NULL)
18   {
19     /* pass the input data to the Fortran routine */
20     sinusfortran_(&length, indata);
21   }

22   /* set back the transformed data */
23   dplSetScalarsFromCArray (pd, indata, length, "temperature");
24 }
```

The code for Fortran function is as follows:

```
      SUBROUTINE SINUSFORTRAN(N, A)
      DOUBLE PRECISION A(*)
      DO 10 I=1,N
         A(I) = SIN(A(I))
10    CONTINUE
      END
```

The call of Fortran function in line 20 depends on the combination of C/C++ and Fortran compilers so the appropriate compiler documentation should be consulted for each particular compiler.

The example above uses the following helper functions:

```
double *dplGetScalarsCArray(vtkUnstructuredGrid *,
                            const char *, int *);
int dplSetScalarsFromCArray(vtkUnstructuredGrid *, double *,
                            int, const char* );
```

The first of them takes point data object and the name of scalar field and allocates an array and then copies the values of the scalar field to the array. It returns the length

of allocate array as through the last argument.

The second function does the reverse - it takes the array and sets the named points attribute to the array values.

4 Programmable filters in C++

We will start the discussion on `vtkProgrammableFilter` class by an example of using programmable filter in a C++ code. Though use of C++ complicates slightly the example, it allows to show some internal working of the class, which understanding is necessary when trying to use `vtkProgrammableFilter` in Tcl.

4.1 Simple pipeline example

To simplify our discussion and the example we present a program in which visualization pipeline has been reduced to minimum and consists of only two objects:

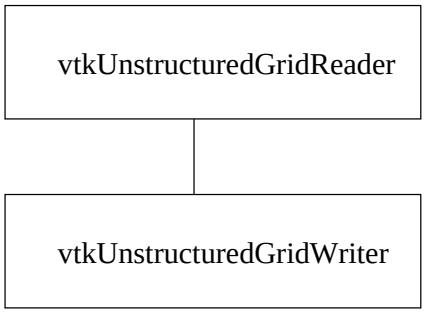

Figure 1: Simple pipeline

The program looks like follows

```
1  #include "vtkUnstructuredGridReader.h"
2  #include "vtkUnstructuredGridWriter.h"

3  int main()
4  {
5     vtkUnstructuredGridReader *reader;
6     vtkUnstructuredGridWriter *writer;

7     reader = vtkUnstructuredGridReader::New();
8       reader->SetFileName("2Dmesh.vtk");

9     writer = vtkUnstructuredGridWriter::New();
10      writer->SetFileName("newMesh.vtk");

11    // connect object to form the pipe
12    writer->SetInput(reader->GetOutput());

13    // initialize pipe processing
```

```
14    writer->Update();

15    writer->Delete();
16    reader->Delete();
17    return 0;
18 }
```

As it can be seen, this program does nothing else as copying "2Dmesh.vtk" file into "newMesh.vtk".

4.2 Pipeline with filter

Now we introduce programmable filter in order to transform an unstructured grid by user specified function. The layout of the program is shown in the figure below. We

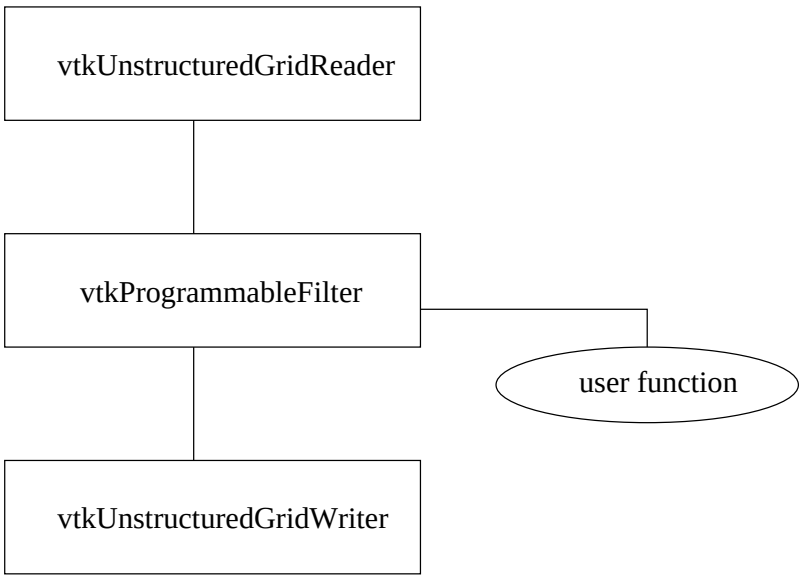

Figure 2: Pipeline with filter

assume that data file specified a scalar attribute for each point. The user function copy the grid topology and geometry and sets new point attributes which are the old value multiplied by 10. With user function as above it would be better to use `vtkProgrammableAttributeDataFilter` but we will use `vtkProgrammableFilter` to show how to create new output object and how to copy grid topology and geometry.

The `vtkProgrammableFilter` class provides, among others, the following method:

```
vktProgrammableFilter::SetExecuteMethod (void(*f)(void *), void *arg)
```

This method takes two arguments: first being the pointer to user function. The user function must take one argument of void pointer type and return void. The second argument is the pointer to client data, which will be passed to user function upon its execution. The client data allow to pass to user function all necessary information the

function needs to perform its job. The client data will usually contain pointer to the filter itself which allows the function to retrieve filter's input and output objects. In the simplest case the client data will be the pointer to the filter alone.

Lets assume that the function

```
void ScaleBy10 (void *arg);
```

is going to be used by the filter. Here is the new program:

```
1  #include "vtkUnstructuredGridReader.h"
2  #include "vtkUnstructuredGridWriter.h"
3  #include "vtkProgrammableFilter.h"

4  void ScaleBy10 (void *arg);

5  int main()
6  {
7     vtkUnstructuredGridReader *reader;
8     vtkUnstructuredGridWriter *writer;
9     vtkProgrammableFilter *filter;

10    reader = vtkUnstructuredGridReader::New();
11      reader->SetFileName("2Dmesh.vtk");

12    writer = vtkUnstructuredGridWriter::New();
13      writer->SetFileName("newMesh.vtk");

14    filter = vtkProgrammableFilter::New();
15      filter->SetExecuteMethod (ScaleBy10, (void*)filter);

16    // connect objects to form the pipe
17    filter->SetInput(reader->GetOutput()):
18    writer->SetInput(filter->GetUnstructuredGridOutput());

19    // initialize pipe processing
20    writer->Update();

21    writer->Delete();
22    filter->Delete
23    reader->Delete();
24    return 0;
25 }
```

Note the line 15 where as we said we pass pointer to filter object as a client data.

4.3 User function

The task of the user function `ScaleBy10` create new grid with the same geometry and topology as the input grid but with points attribute scaled by 10. First the topology and geometry is copied from the input object. Then point data array are copied and

then modified. At the end the modified data array is inserted into output mesh as the new point data. The code for user function is given below.

```
1  #include "vtkDataSet.h"
2  #include "vtkDoubleArray.h"
3  #include "vtkDataArray.h"
4  #include "vtkProgrammableFilter.h"
5  #include "vtkUnstructuredGrid.h"

6  void ScaleBy10(void *arg)
7  {
8    vtkIdType numPts;
9    vtkDataArray *da;
10   vtkDoubleArray *scalars;
11   vtkDataSet *input;
12   vtkUnstructuredGrid *output;

13   vtkProgrammableFilter *myFilter;

14   /* get the filter from client data */
15   myFilter = (vtkProgrammableFilter *)arg;

16   /* get the filter input and output */
17   input = myFilter->GetInput();
18   output = myFilter->GetUnstructuredGridOutput();

19   /* copy grid geometry and topology */
20   output->CopyStructure(input);

21   /* copy the scalar point attribute */
22   scalars = vtkDoubleArray::New();
23   scalars->DeepCopy((input->GetPointData())->GetScalars());

24   /* modify the scalars */
25   double factor = 10;

26   for (vtkIdType j=0; j<scalars->GetNumberOfTuples(); j++)
27   {
28     scalars->SetTuple1(j, factor*scalars->GetTuple1(j));
29   }

30   /* Set back the scalars as the point data in output object */
31   (output->GetPointData())->SetScalars(scalars);
32 }
```

In the line 15 we retrieve the pointer to the filter from client data. In lines 17 to 20 we get the filter input and output and do the copying of grid geometry and topology. Lines 22 to 29 contain the code to copy point scalar attributes into array of doubles and to modify the array. Finally in line 31 we set the modified array as the point data in output object.

The action in lines 26 to 29 is the core of the filter. We could substitute those lines by a call to Fortran routine which does the multiplication. Of course, in light of what we said about passing data from C++ objects to Fortran code, we would need to add appropriate data transfer routines.

5 Programmable filters in Tcl

Now we are about to repeat the example from the previous section entirely in Tcl.

5.1 Simple pipe example

Here is the Tcl code that do not use any filter.

```
1 package require vtk

2 vtkUnstructuredGridReader reader
3    reader SetFileName "2Dmesh.vtk"

4 vtkUnstructuredGridWriter writer
5    writer SetFileName "newMesh.vtk"

6 writer SetInput [reader GetOutput]

7 writer Write

8 vtkCommand DeleteAllObjects
9 exit
```

5.2 Pipeline with filter

Next we present the code in which programmable filter is used to modify the data.

```
1 package require vtk

2 proc ScaleBy10 {} {
3   set input [filter GetInput]
4   set output [filter GetUnstructuredGridOutput]
5   #copy the geometry and topology
6   $output CopyStructure $input

7   set dai [[$input GetPointData] GetScalars]

8   # copy the point scalar attributes to scalars array
9   vtkDoubleArray scalars
10  scalars DeepCopy $dai

11  # modify the array values
12  set n [scalars GetNumberOfTuples]
13  set factor 10
14  for {set i 0} {$i < $n} {incr i} {
15     scalars SetTuple1 $i [expr $factor * [scalars GetTuple1 $i]]
```

```
16   }

17   # get the scalars array as the point data in the output mesh
18   [$output GetPointData] SetScalars scalars
19 }

20 vtkUnstructuredGridReader reader
21    reader SetFileName "2Dmesh.vtk"

22 vtkUnstructuredGridWriter writer
23    writer SetFileName "newMesh.vtk"

24 vtkProgrammableFilter filter
25    filter SetInput [reader GetOutput]
26    filter SetExecuteMethod ScaleBy10

27 writer SetInput [filter GetUnstructuredGridOutput]
28 writer Write

29 vtkCommand DeleteAllObjects

30 exit
```

The example looks similar to the C++ one though there is one important difference – the filter method `SetExecuteMethod` was called with only one argument:

```
filter SetExecuteMethod ScaleBy10
```

It should be called with only one argument because the second argument is implicitly provided by the Tcl wrapper of `SetExecuteMethod` and this argument is set to be the pointer to Tcl interpreter. `vtkProgrammableFilter` filter object need that pointer to evaluate the user supplied Tcl script. The above brings the question how user supplied script will get its client data and in its filter. To solve this problem we must use some other data passing mechanism available in Tcl. We can pass client data through global variable or through use of `namespace variable` mechanism. In presented example we used the fact, that the filter was created in the global name scope and the command procedure associated with it is available everywhere.

6 Using dynamically linked function as filter method in Tcl

In the example of the previous section the user function `ScaleBy10` was written in scripting language. What we would like to do now is to keep the previous example intact except for the user function, which is going to be written in C, C++, or Fortran. We may want to do it for several reasons; first for efficiency, second we may already have substantial portion of the user function code written in another language.

Before we attempt to give a recipe how to achieve the above goal some introduction to extending Tcl applications with new built-in commands in necessary. This topic is wide and of course we are not going to cover it in every detail. More detailed discussion can be found in chapter 44 of [3] or in part III of [4]. It also will be necessary to understand some inner working of VTK library. Unfortunately for that we must refer to the source

code itself. In particular we will refer to the following files: Common/vtkTclUtil.h Common/vtkTclUtil.cxx, Graphics/vtkProgrammableFilter.cxx, Graphics/vtkProgrammableFilterTcl.cxx.

6.1 Adding new built-in command to Tcl interpreter

Tcl may be easily extended by writing new command implementations in C.[1] New commands can be implemented in C for efficiency or to provide functionality which is not possible to provide in pure Tcl.

The example below provides implementation of `Add` command, which adds two numbers and returns the result of addition. Here is how Tcl implementation may look like:

```
proc Add {a b} {
    return [expr $a + $b]
}
```

Instead of the Tcl code we would like to use the following C code:

```
double AddVeryFast(double a, double b)
{
  return (a+b);
}
```

To implement Tcl command in C we must provide a C function called *command procedure*. In our example we will use command procedure interface which is based on so called "dual ported objects". Thews "objects" are structures of type `Tcl_Obj` and were introduced in Tcl 8.0 to minimize number of conversions between string and native representation of data. Each command procedure has standard interface. For our Add command it looks like:

```
int AddObjCmd(ClientData clientData, Tcl_Interp *interp,
                          int objc, Tcl_Obj *CONST objv[]);
```

The first argument allows to pass arbitrary client data to command procedure. The second is the Tcl interpreter, the third the number of arguments passed to the command (including command name) and the fourth is the array of argument values represented as `Tcl_Obj` "objects". The function returns an integer indicating success or failure. Here is an implementation of `AddObjCommand`

```
1 #include <tcl.h>

2 int AddObjCmd(ClientData clientData, Tcl_Interp *interp,
3                          int objc, Tcl_Obj *CONST objv[])
4 {
5   double a, b, c;
6   static Tcl_Obj *resultObjPtr=NULL;

7   /* check the number of arguments: command a b */
```

[1] In this section whenever we say 'C' we also mean 'C++'

```
8   if (objc != 3)
9   {
10    Tcl_WrongNumArgs(interp, 1, objv,
11                     "arg1 arg2\n\t arg1,arg2 are numeric values");
12    return TCL_ERROR;
13  }
14  /* get the value of the first argument */
15  if (Tcl_GetDoubleFromObj(interp, objv[1],&a) != TCL_OK)
16  {
17     return TCL_ERROR;
18  }
19  /* get the value of the second argument */
20  if (Tcl_GetDoubleFromObj(interp, objv[2], &b) != TCL_OK)
21  {
22     return TCL_ERROR;
23  }
24  /* create the result object with value being the sum of arguments */
25  c = AddVeryFast(a, b);
26  resultObjPtr = Tcl_NewDoubleObj(c);
27  Tcl_SetObjResult(interp, resultObjPtr);

28  return TCL_OK;
29 }
```

In lines 8 to 13 we check if command was called with proper number of arguments. In lines 15 to 23 we get the value of the objects which were passed as command arguments. In line 25 we call our C "fast addition" function and in line 26 we create the result object then set it as the result of the command.

The example only touches the issue of creating command procedures in C. The Tcl API (Application Programmer Interface) provides several functions to assist writing command procedures (support for lists, hash tables, script evaluation, variables tracking, etc.).

Creating a command procedure is only half of the job. Tcl must be informed about the new command, so that whenever it encounters the command name in a procedure call context it knows what to do. We are going to provide new built-in Tcl `Add` command as a dynamically loadable package. A package can be loaded with the following command:

```
load library package
```

The first argument to `load` command is the name of a shared library file (usually named *.dll on Windows and *.so on Unix). The *package* is the name of the package and this name is used to call package initialization function. Each package must provide initialization function called *package*_Init. The initialization function performs all necessary initialization and in particular register new commands provided by the package. The initialization function for our example is given below:

```
1 #include <tcl.h>

2 int AddObjCmd(ClientData clientData, Tcl_Interp *interp,
```

```
3                           int objc, Tcl_Obj *CONST objv[]);

4  extern "C" {
5    int Add_Init(Tcl_Interp *interp) ;
6  } /* end extern "C" */

7  int Add_Init(Tcl_Interp *interp)
8  {
9    /* Register the Add command */
10   Tcl_CreateObjCommand(interp, "Add", AddObjCmd, (ClientData) NULL,
11                        (Tcl_CmdDeleteProc *)NULL);

12   /* Declare the package */
13   Tcl_PkgProvide(interp, "Add", "1.1");

14   return TCL_OK;
15 }
```

In lines 4 to 6 we declare `Add_Init` as `extern "C"` to prevent C++ compiler to mangle its name otherwise the `load` command could not find the initialization procedure.

Now, we can create the shared library (.dll for windows or .so for unix) for the package by compiling the *AddObjCmd* and *Add_Init* functions files. The use of `Add` package is demonstrated under unix by the following Tcl script

```
load ./add.so Add
set c [Add 1 2]
puts $c
```

6.2 Tcl interface to VTK library

Though the details might be different the Tcl interface to VTK library is created in a way similar to the one presented in the previous section. For each VTK class the VTK library provides a command procedure and initialization procedure. The initialization procedure (or procedures) are called when VTK library is loaded into Tcl, i.e. when the interpreter encounters lines like:

```
package require vtk
```

The command procedure for a given class creates new class instances (objects) and new commands specific to each instance. In the following Tcl script

```
vtkUnstructuredGridReader myreader
    myreader SetFileName "mesh.vtk"
```

two things happen. First when interpreter encounters command `vtkUnstructuredGridReader` it calls this command procedure provided by VTK library. That procedure creates in turn an object of `vtkUnstructuredGridReader` type and registers new Tcl command called `myreader`. The `myreader` command will then deal with all requests to the `vtkUnstructuredGridReader` object - it will forward the

requests and their arguments to the object by calling object methods. If we create another reader by calling `vtkUnstructuredGridReader myreader1` new Tcl command `myreader1` will be registered and so on. It is important to remember that each VTK object is related to its unique Tcl command, and the command name determines the specific object.

Figure 3 illustrates the way of accessing object methods from Tcl script.

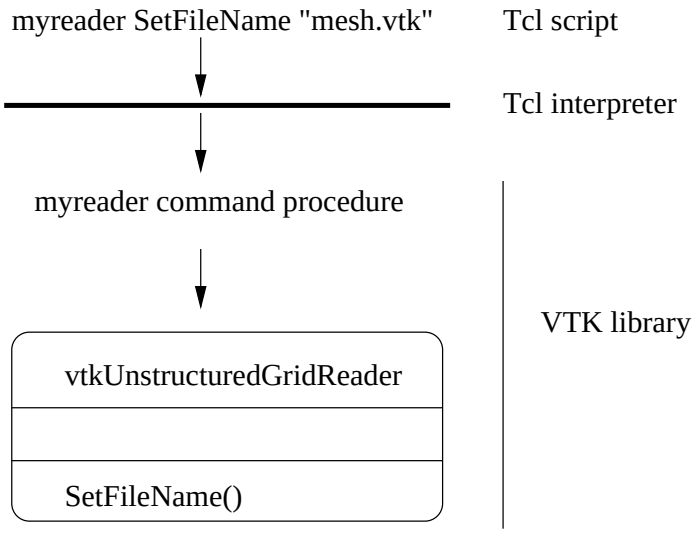

Figure 3: Accessing object methods from Tcl

6.3 Accessing C++ object from Tcl script

Imagine that we have C++ function which takes a pointer to an unstructured grid object. We have wrapped that function in Tcl (by hand like in section 6.1 or using tools like SWIG or CABLE) and now we are about to call that function. However we do not have the right data to pass to it. When in Tcl shell we issues a command

```
vtkUnstructuredGrid mygrid
```

a `vtkUnstructuredGrid` object is created together with its command procedure. What we deal with in Tcl is the object command and not the object pointer. The C++ and Tcl layers are completely separated. VTK however provides utilities in `vtkTclUtil.cxx` which allow to access object pointer knowing its command name and its class name. It also provides reverse function to create object command from object pointer but here we will discuss and use only the former.

With help of the function

```
extern VTKTCL_EXPORT void *
vtkTclGetPointerFromObject(const char *name, const char *result_type,
                           Tcl_Interp *interp, int &error);
```

it is possible to write a new Tcl command which will retrieve the pointer to object and return it to scripting language (e.g. as an integer value, or string value) where it can be passed around.

6.4 `vtkProgrammableFilter` with C++ user function

When trying to use user function written in Fortran we will encounter two problems – accessing data of VTK objects from Fortran and accessing C++ representation of objects from Tcl. The former problem was discussed in section 3 so now we will only concentrate on the latter.

Lets assume that in Tcl we created a programmable filter and we would like to call C++ function

```
void ScaleBy10 (void *arg);
```

as its method. We have to solve two additional problems. First is how to get the pointer to filter object and pass it to the user function as a client data and the second how to call the C++ function.

As for accessing the pointer we can use `vtkTclGetPointerFromObject`. For the second problem we must recall that in Tcl the argument to `SetExecuteMethod` is the Tcl script. When the filter executes the script is evaluated in the context of the current interpreter. What we have to do is to create new special Tcl command which we can pass to the Tcl interpreter and this command will take care to get the pointer to filter object and to call the C++ function. Lets call the Tcl command `ScaleBy10`. When this command is called from Tcl it calls the C++ function `ScaleBy10(void*)`. Additionally the Tcl command is able to register the name of a programmable filter with which it is going to be used.

Now we repeat the example from section 5.2 but this time filter method is implemented in C++.

```
 1 package require vtk

 2 load ./ScaleBy10  ScaleBy10

 3 vtkUnstructuredGridReader reader
 4    reader SetFileName "2Dmesh.vtk"

 5 vtkUnstructuredGridWriter writer
 6    writer SetFileName "newMesh.vtk"

 7 vtkProgrammableFilter filter

 8    # register the filter in callback routine
 9    ScaleBy10 set filter

10    filter SetInput [reader GetOutput]
11    filter SetExecuteMethod ScaleBy10

12 writer SetInput [filter GetUnstructuredGridOutput]
13 writer Write
```

```
14 vtkCommand DeleteAllObjects

15 exit
```

The complete code for ScaleBy10 package is given below. The code has the same structure as the one discussed in 6.1.

```
#include <tcl.h>
#include "string.h"
#include "ScaleBy10.h"
#include "vtkTclUtil.h"

extern "C" {
int ScaleBy10ObjCmd(ClientData clientData, Tcl_Interp *interp,
                            int objc, Tcl_Obj *CONST objv[]);
int Scaleby10_Init(Tcl_Interp *interp) ;
}

int Scaleby10_Init(Tcl_Interp *interp)
{
  /* Initialize the stub table interface */
  if (Tcl_InitStubs(interp, "8.1", 0) == NULL)
  {
     return TCL_ERROR;
  }
  /* Register the command */
  Tcl_CreateObjCommand(interp, "ScaleBy10", PscalarsObjCmd,
  (ClientData) NULL, (Tcl_CmdDeleteProc *)NULL);

  /* Declare the package */
  Tcl_PkgProvide(interp, "ScaleBy10", "1.1");

  return TCL_OK;
}
```

The code above registers new command with the interpreter. It is exactly the same as in the example with `Add` command in section 6.1. Here comes the function which actually implements the command:

```
int ScaleBy10ObjCmd(ClientData clientData, Tcl_Interp *interp,
                            int objc, Tcl_Obj *CONST objv[])
{
  void *filterPtr = NULL;
  char *filterName = NULL;
  int index;
  static Tcl_Obj *filterNameObjPtr=NULL;
  Tcl_Obj *objPtr;

  /* array of subcommand names */
  char *subCmds[] = "set", "get", NULL;
```

```
enum CmdIdx {SetIdx, GetIdx };

/* check if command was called with proper number of arguments */
if (objc > 3)
{
  Tcl_WrongNumArgs(interp, 1, objv, "?get?, ?set FilterName?");
  return TCL_ERROR;
}

/* If no argument was given then try to call associated C++ function */
if (objc == 1)
{
   /* check if we have object which holds filter name */
   if (filterNameObjPtr == NULL)
   {
     Tcl_AppendResult(interp, "Error: filter name was not set\n", NULL);
     Tcl_AppendResult(interp, "Use: ", Tcl_GetString(objv[0]),
                                 " set arg",NULL);
     return TCL_ERROR;
   }

   /* get the name from object */
   filterName = Tcl_GetString(filterNameObjPtr);

   int error=0;

   /* Get the pointer to the filter object
      This is the crucial part of the function - getting
      the VTK object knowing the name of its command procedure
    */

   filterPtr = vtkTclGetPointerFromObject(filterName,
                   "vtkProgrammableFilter", interp, error);

   if (error == 1 || filterPtr == NULL)
   {
     Tcl_AppendResult(interp, "Could not fild the vtkProgrammableFilter",
                      " with name: ", filterName, NULL);
     return TCL_ERROR;
   }
   /* Run the filter method */
   ScaleBy10(filterPtr);

   return TCL_OK;
}
else
/* if more arguments are given then check if they are valid subcommands */
{
   if (Tcl_GetIndexFromObj(interp, objv[1], subCmds, "subcommand",
       TCL_EXACT, &index) != TCL_OK)
   {
      return TCL_ERROR;
```

```
    }
    /* check if subcommands were called with proper number of arguments */
    if (  (index == SetIdx && objc != 3)
        ||(index == GetIdx && objc != 2))
    {
      Tcl_WrongNumArgs(interp, 1, objv, "?get?, ?set FilterName?");
      return TCL_ERROR;
    }

    /* execute the particular subcommand */
    switch (index)
    {
      /* duplicate the argument given to 'set' subcommand
         and store it as filter name
       */
      case SetIdx:
         if (filterNameObjPtr != NULL)
         {
           Tcl_DecrRefCount(filterNameObjPtr);
         }
         filterNameObjPtr = Tcl_DuplicateObj(objv[2]);
         Tcl_IncrRefCount(filterNameObjPtr);
      /* return the stored filter name */
      case GetIdx:
         objPtr = Tcl_DuplicateObj(filterNameObjPtr);
         /* Setting the results add new reference so after
          * the reference count is 1
          */
         Tcl_SetObjResult(interp, objPtr);
         return TCL_OK;
      default: /* just sanity check */
         char errMsg[256];
         snprintf(errMsg, 255, "index value %d, file: %s, line %d",
                          index, __FILE__, __LINE__);
         Tcl_AppendResult(interp, "Internal error, unknown ", errMsg, NULL);
         return TCL_ERROR;
    }
  }
}
```

The command can be called with no arguments - that triggers the execution of the C++ function `ScaleBy10(void *)` or with the following subcommands

- `ScaleBy10 get` – it returns the filter name registered with command
- `ScaleBy10 set` *name* – it register the *name* in the command

The most important it the above code is the line

```
filterPtr = vtkTclGetPointerFromObject(filterName,
            "vtkProgrammableFilter", interp, error);
```

In this line we retrieve the pointer to registered filter from the pointer name.

In the presented implementation there is no way to check if the C++ function `ScaleBy10` have finished successfully. To provide such feedback it would be necessary to create compound data object and pass it as a client data. In our case the declaration of such compound object could look like:

```
typedef struct MyClientData {
   void *arg;
   int status;
} MyClientData;
```

If we have more than one C++ function to use as a filter method we need to provide appropriate Tcl command for each of them. As the number of function increases the task of writing the command procedure becomes tedious. However if we look at given example we may notice that the implementation of command procedure will be *exactly* the same for all functions except the names. It is very easy to provide a script which will generate the implementations from the list of function names.[2]

References

[1] W. Schroeder, K. Martin, B. Lorensen: The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition. Kitware, Inc. 2003

[2] The VTK User's Guide. Edited by W. Shroeder. Kitware, Inc. 2003

[3] B.B. Welch: Practical programming in Tcl and Tk. 3rd edition, Prentice-Hall, 1999.

[4] J.K. Ousterhout: Tcl and the Tk Toolkit. Professional Computing Series. Addison-Wesley, 1994.

[2] One may ask if it is possible to use for instance SWIG to to that job. It is possible however not necessary and in fact it would require somehow to twist normal SWIG behaviour. It is much simpler and appropriate to use bash, AWK or sed script

A Example of accessing object data with routines from dpl library

```
1  #include <iostream>
2  #include <assert.h>
3  #include "dpl.h"

4  using namespace std;

5  template <typename T> void PrintArray( T *array, int length, int stride)
6  {
7    T *pt;
8    pt = array;
9    for (int i=0; i<length; i++)
10   {
11     cout << i << " : " ;
12     for (int j=0; j<stride; j++)
13     {
14        cout << "  " << *pt;
15        pt++;
16     }
17     cout << endl;
18   }
19 } /* end of PrintArray */


20 int main( int argc, char *argv[] )
21 {
22   vtkIdType length;
23   double coords[12] = { 0, 0, 0,
24                         1, 0, 0,
25                         1, 1, 0,
26                         0, 1, 0};
27   int ids[6] = {0, 1, 2,
28                 0, 2, 3};

29   double temperature[4] = {1, -2, 3, 4};
30   double density[4] = {0.1, 0.2, 0.12, 0.23};

31   double *rcoords;
32   int *rids;
33   double *rscalars;

34   /* Create unstructured grid from arrays */
35   vtkUnstructuredGrid *ugrid = dplUGridTriangleFromCArrays (4, coords, 2, ids);

36   vtkPointData *pd = ugrid->GetPointData();

37   /* set the scalar fileld in points */
38   dplSetScalarsFromCArray(pd, temperature, 4, "temperature");

39   /* Get coordinates array */
```

```
40   rcoords = dplGetCoordCArray (ugrid->GetPoints(),&length);
41   assert (length == 12);
42   cout << "Coordinates : " << endl;
43   PrintArray (rcoords, 4, 3);

44   /* Get indices */
45   rids = dplGetTriangleIdsCArray (ugrid, &length);
46   assert (length == 6);
47   cout << "Element nodes : " << endl;
48   PrintArray (rids, 2, 3);

49   /* get point data */
50   rscalars = dplGetScalarsCArray (pd, "temperature", &length);
51   assert (length == 4);
52   cout << "Points scalar attribute : " << endl;
53   PrintArray (rscalars, 4, 1);

54   delete [] rscalars;

55   /* set point data */
56   dplSetScalarsFromCArray(pd, density, 4, "density");
57   rscalars = dplGetScalarsCArray (pd, "density", &length);
58   assert (length == 4);
59   cout << "Points scalar attribute : " << endl;
60   PrintArray (rscalars, 4, 1);

61   ugrid->Delete();
62   delete [] rids;
63   delete [] rcoords;
64   delete [] rscalars;

65   return 0;
66 } /* end of main */
```

B dpl library

B.1 Getting points coordinates

```
1  /* NAME
2   *   dplGetCoordCArray
3   * DESCRIPTION
4   *   Allocates an array and fill it with point coordinates (3D).
5   * ARGUMENTS
6   *    points - the vtkPoints object (IN)
7   *    length - variable used to return the length of allocated array
8   *             If NULL then length is not set. (OUT)
9   * RETURN VALUE
10  *    Pointer to allocated array or NULL if allocation failed.
11  */
12 double * dplGetCoordCArray (vtkPoints *points, vtkIdType *length)
13 {
14   /* get the number of points */
15   vtkIdType n  = points->GetNumberOfPoints();

16   /* set the length of created array */
17   if (length != NULL)
18   {
19     *length = 3 * n;
20   }
21   double *dcoords = new double[3 * n];
22   if (dcoords != NULL)
23   {
24     double *dcp = dcoords;
25     // Copy point coordinates to dcoords array
26     for (vtkIdType i=0; i<n; i++)
27     {
28       points->GetPoint(i, dcp);
29       dcp += 3;
30     }
31   }
32   return dcoords;
33 } /* end of dplGetCoordCArray */
```

B.2 Setting point coordinates

```
/* NAME
 *   dplSetCoordsFromCArray
 * DESCRIPTION
 *   Sets points coordinates from given array.
 *   The points object must already have enough points.
 * ARGUMENTS
 *    dcoords - continuous array of n tuples. Each tuple has three
 *              components x,y,z. (IN)
 *    n       - number of tuples in the dcoords array (IN)
```

```
 *    points  - vtkPoints object
 * RETURN VALUE
 *    1 - if successful
 *    0 - if vtkPoints has not enough points
 *   -1 - if vtkPoints has more points than tuples in the dcoords array.
 * NOTE
 *    In case of 0 or -1 the point coordinates are not changed
 */
int dplSetCoordsFromCArray (const double *dcoords, const vtkIdType n,
                            vtkPoints *points)
{
  /* get the number of points */
  vtkIdType np  = points->GetNumberOfPoints();

  if (np == n)
  {
    double *dcp = const_cast<double *>(dcoords);
    // set point coordinates
    for (vtkIdType i=0; i<np; i++)
    {
      points->SetPoint(i, dcp);
      dcp += 3;
    }
    return 1;
  }
  else if (np < n )
  {
     return 0;
  }
  else /* np > n */
  {
     return -1;
  }
} /* end of dplSetCoordsFromCArray */
```

B.3 Getting indices of nodes of triangular elements

```
/* NAME
 *   dplGetTriangleIdsCArray
 * DESCRIPTION
 *   Allocates and return an array of indices of VTK_TRIANGLE cells.
 * ARGUMENTS
 *   ugrid - vtkUnstructuredGrid object (IN)
 *   length - variable used to return the length of allocated array
 *            (3*number of cells). If NULL then length is not set. (OUT)
 * RETURN VALUE
 *   Pointer to allocated array or NULL if allocation failed or
 *   grid does not have VTK_TRIANGLE cells.
 */
vtkIdType * dplGetTriangleIdsCArray (vtkUnstructuredGrid *ugrid,
```

```
                                        vtkIdType *length)
{
  assert (ugrid != NULL);

  /* get the array of indices of cells of VTK_TRIANGLE TYPE */
  /* Here we assume that grid can consist of cell of different types */
  vtkIntArray *triangleIds  = vtkIntArray::New();
  ugrid->GetIdsOfCellsOfType (VTK_TRIANGLE, triangleIds);

  /* get number of triangles */
  vtkIdType nt = triangleIds->GetNumberOfTuples();

  /* set the array length */
  if (length != NULL)
  {
    *length = 3 * nt;
  }

  /* allocate array for triangle indices */
  vtkIdType *idsarray = new vtkIdType [nt * 3];

  if (idsarray != NULL)
  {
    vtkIdType *id = idsarray; /* pointer to point at idsarray elements */
    vtkIdList *idlist = vtkIdList::New();
    /* for each triangular element */
    for (vtkIdType i=0; i<nt; i++)
    {
      /* get the id of the VTK_TRIANGLE cell */
      vtkIdType triangId = triangleIds->GetValue(i);

      /* get the list of id of cell points */
      ugrid->GetCellPoints(triangId, idlist);

      /* copy the id of cell points of the idsarray */
      for(int j=0; j<3; j++)
      {
        *id = idlist->GetId(j);
         id+=1;
      }
    }
    idlist->Delete();
  }

  triangleIds->Delete();

  return idsarray ;
} /* end of dplGetTriangleIdsCArray */
```

B.4 Setting scalar points attribute

```
/* NAME
 *    dplSetScalarsFromCArray
 * DESCRIPTION
 *    This function set the SCALARS attributes at points. The attributes
 *    are copied from given array. The name argument specifies which
 *    scalar arguments to set. If there is no scalar arguments of given
 *    name then the new scalar attribute is created, added to points data
 *    and set as the active scalar attribute.
 * ARGUMENTS
 *    pd  - point data object
 *    sdata - array of doubles containing scalar data
 *    n     - the length of sdata array
 *    name  - the name of attributes
 * RETURN VALUE
 *    1 - if successful
 *    0 - if number of elements in sdata is not the same as number of points
 *    -1 - if points have attribute of given name but is not of type double
 *   -2 - if cannot add new scalar attribute because all scalar attribute
 *        slots are engaged.
 */
int dplSetScalarsFromCArray (vtkPointData *pd,
                             const double *sdata,
                             vtkIdType n,
                             const char *name)
{
   int ierr;

   assert (pd != NULL);

   /* Set the active scalar attribute */
   int hasScalars = pd->SetActiveScalars(name);

   if (hasScalars != -1 )
   {
     /* set the scalars value */
     ierr = dplSetScalars(pd, sdata, n);
   }
   else
   {
     /* create new scalars data and add it to point data */
     ierr = dplAddScalars(pd, sdata, n, name);
   }
   return ierr;
} /* dplSetScalarsFromCArray */

/* helper function to set scalars */
static int dplSetScalars (vtkPointData *pd, const double *sdata,
                             const vtkIdType n)
{
```

```
  vtkDataArray *da = pd->GetAttribute(vtkDataSetAttributes::SCALARS);
  if (da->GetDataType() != VTK_DOUBLE)
  {
    return -1;
  }
  if (da->GetNumberOfTuples() != n)
  {
    return 0;
  }
  /* just sanity check */
  assert (da->GetNumberOfComponents() == 1);

  /* set the array */
  for (vtkIdType j=0; j<n; j++)
  {
    da->SetTuple1(j, sdata[j]);
  }
  return 1;
} /* end of dplSetScalars */

/* helper function to add scalar array to point data */
static int dplAddScalars (vtkPointData *pd, const double *sdata,
                          const vtkIdType n, const char *name)
{
  vtkDoubleArray *da = vtkDoubleArray::New();

  da->SetName(name);
  da->SetNumberOfComponents(1);
  da->Allocate(n);
  da->SetNumberOfTuples(n);

  /* set the array */
  for (vtkIdType j=0; j<n; j++)
  {
    da->SetTuple1(j, sdata[j]);
  }

  /* add array to point data */
  pd->AddArray(da);

  int hasScalar = pd->SetActiveScalars(name);
  assert (hasScalar != -1);

  return 1;
} /* end of dplAddScalars */
```

B.5 Getting scalar points attribute

```
/* NAME
 *   dplGetScalarsCArray
```

```
 * DESCRIPTION
 *   Allocate and fill the array with point scalar data
 * ARGUMENTS
 *   pd - point data object (IN)
 *   name - name of the scalars (IN)
 *   length - variable used to return the length of allocated array
 * RETURN VALUE
 *   Pointer to allocated array or NULL if there was no scalar attribute
 *   of given name or array allocation failed
 */
double * dplGetScalarsCArray (vtkPointData *pd, const char *name,
                              vtkIdType *length)
{
   assert (pd != NULL);

   /* Set the active scalar attribute */
   int hasScalar = pd->SetActiveScalars(name);
   if (hasScalar == -1)
   {
     return NULL;
   }
   vtkDataArray *da = pd->GetAttribute(vtkDataSetAttributes::SCALARS);

   vtkIdType n = da->GetNumberOfTuples();

   if (length != NULL)
   {
     *length = n;
   }

   /* we are expecting a scalar date with one component only */
   assert (da->GetNumberOfComponents() == 1);

   double *dscalars = new double [n];

   if (dscalars == NULL)
     return NULL;

   /* set the array */
   for (vtkIdType j=0; j<n; j++)
   {
     da->GetTuple(j, dscalars+j);
   }

   return dscalars;
} /* end of dplGetScalarsCArray */
```

B.6 Creating vtkUnstructuredGrid from arrays data

```
/* NAME
```

```
 *   dplUGridTriangleFromCArrays
 * DESCRIPTION
 *   Create an ustructured grid with triangular elements using coordinates
 *   and topology arrays
 * ARGUMENTS
 *   npoints - number of points
 *   dcoords - array of point coordinates (of size 3 * npoints)
 *   ncells - number of triangular cells in the grid
 *   ids - array of indices of cell points
 * RETURN VALUE
 *   Pointer to vtkUnstructured or NULL on error
 */
vtkUnstructuredGrid * dplUGridTriangleFromCArrays (vtkIdType npoints,
                  const double *dcoords, vtkIdType ncells,
                  vtkIdType *ids)
{
  vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
  vtkPoints *points = vtkPoints::New();

  assert (ugrid != NULL);
  assert (points != NULL);

  /* set points coordinates */
  points->SetNumberOfPoints(npoints);
  for (vtkIdType i=0; i<npoints; i++)
  {
     points->SetPoint(i, dcoords+3*i);
  }
  ugrid->SetPoints(points);
  points->Delete();

  /* set cell indices */
  ugrid->Allocate(ncells);
  for (vtkIdType i=0; i<ncells; i++)
  {
    ugrid->InsertNextCell(VTK_TRIANGLE, 3, ids+3*i);
  }

  return ugrid;
} /* end of dplUGridTriangleFromCArrays */
```